Chapter 2: Variables (Objects) and Basic Types

Namespace `using` Declaration

The *namespace* is a mechanism for putting names defined by a library into a single space. *Namespaces* help avoid name clashes (抵觸). The names defined by the C++ library are in the namespace `std`. For example,

```
std::cout
```

A `using` declaration allows us to access a name from a *namespace* without the scope cumbersome prefix (`std`) and the resolution operator (`::`).

```
cout
```

nameSpaceExample1.cpp

```
#include <iostream>

// using declarations for names from the standard library
using std::cin;
using std::cout;
using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;

    int v1, v2;
    cin >> v1 >> v2;

    cout << "The sum of " << v1
          << " and " << v2
          << " is " << v1 + v2 << endl;

    return 0;
}
```

Or usage of the entire `std` namespace:

nameSpaceExample2.cpp

```
#include <iostream>

// using declarations for the entire standard library
using namespace std;

int main()
{
    cout << "Enter two numbers:" << endl;

    int v1, v2;
```

```
    cin >> v1 >> v2;

    cout << "The sum of " << v1
          << " and " << v2
          << " is " << v1 + v2 << endl;

    return 0;
}
```

Inside .h header files, we should *always* use the fully qualified library names, that is, **DO NOT** use the `using` declaration for *namespaces*. Why? You might introduce name clashes for any file including this .h header file.

```
#ifndef COORDH
#define COORDH
using namespace std; // this should be avoided!
                     // any file that includes this hear
                     // will have naming conflicts with std
struct Coord {
    double x;
    double y;
    double z;
    void print_x() {cout << x;}
};
#endif
```

Variables (Objects): A variable provides us with named storage space that our programs can manipulate. C++ programmers tend to refer to variables as "variables" or as "objects" interchangeably. Each variable in C++ has a type. The type determines

- the size and layout of the variable's memory
- the range of values that can be stored within that memory
- the set of operations that can be applied to the variable.

Primitive Built-in Types:

- Integral Types
    - Integers: `short`, `int`, `long`, `long long` --- `signed`, `unsigned`
    - Characters: `char` --- `signed`, `unsigned`
    - Extended Characters: `wchar_t`, `char16_t`, `char32_t`
    - Boolean values: `bool`
    - The `unsigned int` can be abbreviated as `unsigned`.

- Floating-Point Types
    - `float`, `double`, `long double`

Examples

```
int a = 10; // 'int a' is the declaration, and initilized with 10
short i = 1;
double d = 0.5;
long double ld;
```

The size of the type depends on the compiler and machine[1]. There is no strict standard on the sizes for these built-in types; each compiler implementation may specify these sizes that best fit the architecture where the program is going to run. This rather generic size specification for types gives the C++ language a lot of flexibility to be adapted to work optimally in all kinds of platforms, both present and future.

For example, the `int` has at least 16-bit of size (2-bytes), and `long` at least 32-bit (4-bytes).

Use the `sizeof` operator[2] to determine size of the variable type in bytes where code is ran:

```
#include <iostream>
using namespace std;

int main()
{
    cout << sizeof(char)<<"\n";
    cout << sizeof(int)<<"\n";
    cout << sizeof(float)<<"\n";
    cout << sizeof(double)<<"\n";
    return 0;
}
```

Output:

```
1
4
4
8
```

[1] If you are interested, the size of these variables can be found here: URL: http://www.cplusplus.com/doc/tutorial/variables/

[2] More details on `sizeof()` URL: https://www.geeksforgeeks.org/sizeof-operator-c/

Initialization of variables in C++

In C++, initialization of variables is a critical issue. It is somewhat complicated too.

Default initialization. For Plain Old Data (POD) types (also called trivial type), such as `int` and `double`, nothing is done to initialize the variable when declared. This is part of a process called *default initialization*[3]. For example:

```
#include <iostream>

using namespace std;

int main ()
{
    int a,b,c;
    cout << a << endl;
    cout << b << endl;
    cout << c << endl;
}
```

You will see 3 zeros only when you are lucky. There is no guarantee what values you will get. As a result, always initialize your variables with a value is good practice, such as

```
int i = 0;
```

Zero initialization. Alternatively, you can initialize the variable with *zero initialization*[4]:

```
int i = {};
cout << i << endl;
```

Value initialization. For GOD, variables are *zero-initialized* (set to 0) when *value-initialized*[5]. For non-GOD, value initialization has other behaviors. In the 4th syntax in [5] we see that a pair of braces `{}` forces value initialization:

```
int i{};
cout << i;
```

[3] Default initialization URL: https://en.cppreference.com/w/cpp/language/default_initialization

[4] Zero initialization URL: https://en.cppreference.com/w/cpp/language/zero_initialization

[5] Value initialization URL: https://en.cppreference.com/w/cpp/language/value_initialization

This triggers *value initialization* and thus *zero-initialization* for GOD, and `i` will be 0.

We will talk more about these initializations later when we get to objects and classes.

Compound types. A *compound type* (複合型別) is a type defined in terms of another type. We will cover two compound types: *reference* and *pointer*.

2.3.1 Reference (l-value reference[6])

A reference is an alternative name for an object (e.g., 孫文 and 孫中山). An object declared as a reference is merely a second name (alias) assigned to an existing object. No new object is created.

A reference is defined by preceding a variable name by the ampersand (`&`) symbol.

Example

```
#include <iostream>

using namespace std;

int main()
{
    int i = 5;
    int& j = i; // j is an alternative name to i
    int k = i;

    cout << "Before i change, i = " << i << endl;
    cout << "Before i change, j = " << j << endl;
    cout << "Before i change, k = " << k << endl;
    i = 6;
    cout << "After i change: i = " << i << endl;
    cout << "After i change: j = " << j << endl;
    cout << "After i change: k = " << k << endl;
}
```

Example

```
int a = 10;
```

[6] For reference types, you can find info here: URL: https://www.learncpp.com/cpp-tutorial/15-2-rvalue-references/

```
int& r = a; // or (int &r = a;)
r = 20; // assign 20 to the object r refers, i.e., assign to a
Sales_item w;
Sales_item& x = w;
```

Quick Check: What does the following code print?

RefEx.cpp

```
#include <iostream>

using namespace std;

int main()
{
    int i;
    int& ri = i;
    i = 5;
    ri = 10;
    cout << "i = " << i << endl;
    cout << "ri = " << ri << endl;
    return 0;
}
```

A:

```
i = 10
ri = 10
```

Remark: The primary usage of reference is parameter-passing for functions. We will have more to say on the topic later when we talk about functions.

2.3.2 Pointer

A pointer is a compound type that "points to" another type. A pointer holds the memory address of another object[7].

The `*` operator symbol in a variable declaration indicates that the identifier is a pointer.

```
string *pstring;
int* i;
```

When attempting to understand pointer declarations, read them from **right to left**: `pstring` is a pointer that can point to string objects, and `i` is a pointer that points to an `int` object.

[7] More details on pointers and memory can be found: URL: http://www.cplusplus.com/doc/tutorial/pointers/

To retrieve the address of an existing object, use the *address-of* operator (&).

```
string  s("hello world");
string *sp = &s;   //sp holds the address of s
//initialize sp to point to the string named s;
```

Caution: C++ uses & to denote the *address-of* operator in an expression. In declarations, the & declares a reference variable.

We use the *dereference* operator (*) on a pointer to access the object/value.

```
string  s("hello world");
string *sp = &s;   //sp holds the address of s
cout << *sp; // dereference the pointer sp
```

Pointer with reference:

```
int ival = 1024;
int *pi = &ival;       // pi points to an int
int **ppi = &pi;       // ppi points to a pointer to int
cout << "The value of ival" << endl
  << "direct value: " << ival << endl
  << "indirect value: " << *pi << endl
  << "doubly indirect value: " << **ppi << endl << endl;

cout << "raw pi: " << pi << endl
  << "dereferenced pi: " << *pi << endl
  << "raw ppi: " << ppi << endl
  << "dereferenced ppi: " << *ppi << endl;
```

[Attention] Some symbols, such as & and *, are used as both an operator in an expression and as part of a declaration. The context in which a symbol is used determines what the symbol means:

```
int i = 42;
int &r = i; // & follows a type and is part of a declaration;
               // r is a reference
int *p; // * follows a type and is part of a declaration; p is a pointer
p = &i; // & is used in an expression as the address-of operator
*p = i; // * is used in an expression as the dereference operator
int &r2 = *p; // & is part of the declaration; * is the dereference operator
```

<u>Quick Check</u>: What does the following program print?

```
#include <iostream>

using namespace std;

int main()
{
    int i = 4;
    int* pi = &i;
    *pi = *pi * *pi;
    cout << "i = " << i << endl;
    cout << "*pi = " << *pi << endl;
    return 0;
}
```

A:

```
i = 16
*pi = 16
```

Brief Summary: Compound Type

In C++, a type that is defined in terms of another type is called the compound type. We have introduced two compound types so far:

(1) <u>reference</u>: to define an alias for another object; and

(2) <u>pointer</u>: to define an object that can hold the address of an object.

The `const` Qualifier

The `const` qualifier provides a way to transform an object into a constant. For example, we define a constant such as `PI`.

When using constants in programming languages, we must initialize it when it is declared.

```
const double PI = 3.14159265358979323846264338327 95;
```

There are times when the data type is obvious to the compiler given the context. The `auto` and `decltype` provide automatic type inference[8]. We introduce them in the following.

---

[8] Although this is a more advanced topic, more details on automatic type inference can be found URL: https://www.geeksforgeeks.org/type-inference-in-c-auto-and-decltype/

2.5.2 The `auto` Type Specifier

We can let the compiler figure out the type for us by using the `auto` type specifier.

Unlike typical type specifiers, such as `double`, that name a specific type, `auto` tells the compiler to deduce (推斷) the type from the initializer (right hand side value).

```
int ival = 1024;
int *pi = &ival;       // pi points to an int
auto ppi = &pi;     // int **ppi

cout << "val of pi: " << ppi << endl;
cout << "val of *pi: " << *ppi << endl;
cout << "val of **pi: " << **ppi << endl;
```

<u>Reference, `const` and `auto`</u>

The type that the compiler infers for `auto` **is NOT always exactly the same as the initializer's type.** Instead, the compiler adjusts the type to conform to normal initialization rules.

Reference: when we use a **reference**, we are really using the object to which the reference refers. In particular, when we use a reference as an initializer, the initializer is the corresponding object. The compiler uses that object's type for `auto`'s type deduction:

```
int i = 0, &r = i;
auto a = r; // a is an int (r is an alias for i,
                                // which has type int)
```

If we really want the deduced type to have a reference, we must say so explicitly:

```
int i = 0, &r = i;
auto& a = r; // a is now an int&
```

<u>**Top-level** `const`</u>: similarly, `auto` ordinarily ignores top-level `const`. If we really want the deduced type to have a top-level `const`, we must say so explicitly:

```
const int ci = 40;
const auto fi = ci;
```

<u>**Quick Check**</u>: Determine the types of `j`, `k`, `p`, `j2`, `k2` deduced in each of the following definitions.

```
const int i = 42;

auto j = i;
const auto &k = i;
```

```
auto *p = &i;
const auto j2 = i, &k2 = i;
```

A:

```
j // int
k // const int&
p // int*
j2 // const int
k2 // const int&
```

2.5.3 `decltype` Type Specifier

`decltype` tells the compiler to deduce type from an expression. The compiler analyzes the expression to determine its type but does NOT evaluate the expression.

```
int i;
const int ci = 0, &cj = ci;
decltype(ci) x = 0; // x has type const int
decltype (i) a; // a is an uninitialized int
decltype(cj) y = x; // y has type const int& and is bound to x
```

```
double f() {return 3.01;}
decltype(f()) sum = x;
// sum has whatever type f returns, double in this case
```

[Attention] Assignment is an example of an expression that yields a reference type. The type is <u>a reference to the type of the left-hand operand</u>. That is, if `i` is an `int`, then the type of the expression `i = x` is `int&`. Using this knowledge, determine the type of `d` deduced from `decltype` statement

```
int a = 3, b = 4;
decltype(a = b) d = a;
```

A:

```
d // int&
```

Example:

```
int a = 3, b = 4;
decltype(a = b) d = a;
d = 5;
cout << "value of a after change on d: " << a <<endl;
```

Q: What are the outputs?

```
#include <iostream>

using namespace std;
```

```
int main()
{
    int a = 3, b = 4;
    decltype(a) c = a;
    decltype(a = b) d = a;
    c++;
    d += 2;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    cout << "d = " << d << endl;
}
```

A:

```
a = 5
b = 4
c = 4
d = 5
```

Remark: the `decltype` is very useful for template programming as we will realize later in the class. [Attention] The operand of `decltype` does NOT get evaluated!

2.6 Define Our Own Data Structures (or Define Our Own Types)

C++ allows the definition of a new **type**. One way to do this is through the `struct`. The other way to do this is through the `class` which we will cover later.

If the class is defined with the `struct` keyword, then members are `public` if no further access label is imposed. We will go more into details when we get to classes and objects.

Data abstraction (資料抽象化) is a powerful mechanism whereby a set of related objects (often of different types) can be considered or grouped as a single object/type.

For example, we can define a data abstraction `Student` that contains `name` (`string`), `id` (`int`) and `age` (`int`) with the keyword `struct`:

```
struct Student
{
    std::string name;
    int id;
    int age;
};
```

The definition begins with a keyword `struct`, followed by the name of your choice. Then one or more **data members** are declared within curly braces (`{}`). Finally, a semicolon (`;`) concludes the `struct` definition.

(Reflection) Data abstraction allows us to handle data in a more meaningful manner. For example, we now can **think of** `Student` as a new type that can represent a student in the real world. We can then utilize the object of `Student` such as in functions, array or `vector`.

Member initialization: when we create objects, and no constructor was provided, the in-class initializers will be used to initialize the data members. Members without an initializer are *default initialized*.

```
struct Student
{
    std::string name;
    int id = 0;
    int age = 0;
};
```

Q: what does it mean when we define a `Student` object now?

```
Student john;
```

A:

It means we have an object called `john` with a type `Student`, and `john.name` is initilized by the string default constructor, which resulted to an empty string, `john.id` and `john.age` are both initialized to zero.

Here is an example on how to use the `struct`:

```
#include <iostream>
#include <string>

struct Student
{
    std::string name;
    int id = 0;
    int age = 0;
};

int main() {
    Student s;
    s.name = "OOP";
    s.id = 6;
    s.age = 21;
```

```
    std::cout << "Student info: \n";
    std::cout << "\tname: " << s.name << "\n";
    std::cout << "\t  id: " << s.id << "\n";
    std::cout << "\t age: " << s.age << "\n";

    return 0;
}
```

Output:

```
Student info:
    name: OOP
    id: 6
    age: 21
```

2.6.3 Writing Our Own Header Files

Remember how to write headers in lecture 1?

In `Student.h`

```
#ifndef STUDENT_H
#define STUDENT_H
#include <string>
struct Student
{
    std::string name;
    int id = 0;
    int age = 0;
};
#endif
```

And in `main.cpp`

```
#include <iostream>
#include "Student.h"

int main() {
    Student s;
    s.name = "OOP";
    s.id = 6;
    s.age = 21;

    std::cout << "Student info: \n";
    std::cout << "\tname: " << s.name << "\n";
    std::cout << "\t  id: " << s.id << "\n";
    std::cout << "\t age: " << s.age << "\n";

    return 0;
}
```

Here STUDENT_H is the preprocessor variable.

Now if in the main we have include Student.h 2 times. Once directly in the main.cpp, and the other in a.h:

In main.cpp

```
#include "Student.h" // first time
...
#include "a.h" // second time
...
```

In a.h

```
#ifndef A_H
#define A_H
#include "Student.h"
...

#endif
```

Q: what happens when Student.h is included at the first time?

A: The first time Student.h is included, the #ifndef test will succeed. The preprocessor will process the lines following #ifndef up to the #endif. As a result, the preprocessor variable STUDENT_H will be defined and the contents of Student.h will be copied into our program.

Q: what happens when Student.h is included at the second time?

A: If we include Student.h later on in the same file, the #ifndef directive will be false. The lines between it and the #endif directive will be ignored. As a result, no duplication of struct Student.

Chapter 3 Vectors

A vector can store a collection of objects of a single type, each of which has an associated integer index. A vector is a **class template**. It can be used with any data type, built-in or user defined.

What is good about vectors? One important aspect is the arrangement of memory size. In computers, resizing your collection of data can be extremely inefficient. The vector provides a container so that users do not need to worry about memory management.

3.3.1 Defining and Initializing `vector`s

To declare a `vector`, we must supply what type of objects the `vector` will contain. We specify the type by putting it between a pair of angle brackets following the template's name:

```
vector<int> ivec;
vector<Sales_item> salesVec;
vector<vector<int> > matInt;
vector<Student> vs;
```

Table 3.4 The ways to Initialize a `vector`

`vector<T> v1`	`vector` that holds objects of type `T`. Default initialization; `v1` is empty.
`vector<T> v2(v1)`	`v2` has a copy of each element in `v1`.
`vector<T> v2 = v1`	Equivalent to `v2(v1)`, `v2` is a copy of the elements in `v1`.
`vector<T> v3(n, val)`	`v3` has `n` elements with value `val`.
`vector<T> v4(n)`	`v4` has `n` copies of a value-initialized object.
`vector<T> v5{a,b,c...}`	`v5` has as many elements as there are initializers; elements are initialized by corresponding initializers.
`vector<T> v5 = {a,b,c...}`	Equivalent to `v5{a,b,c...}`.

List each element in the following `vector` initialization

```
vector<int> ivec(10, -1);
vector<string> svec(10, "hi");
```

```
vector<int> ivec(10);
vector<int> ivec(10, 1);
vector<int> ivec{10, 1};
```

```
vector<string> svec(10);
vector<Sales_item> salesVec(10);
```

Table 3.5 `vector` Operation

`v.empty()`	Returns `true` if `v` is empty; otherwise returns `false`.
`v.size()`	Returns the number of elements in `v`.
`v.push_back(t)`	Adds an element with value `t` to end of `v`.
`v[n]`	Returns a reference to the element at position `n` in `v`.
`v1 = v2`	Replaces the elements in `v1` with a copy of the elements in `v2`.
`v1 = {a,b,c...}`	Replaces the elements in `v1` with a copy of the elements in the comma-separated list.
`v1 == v2` `v1 != v2`	`v1` and `v2` are equal if they have the same number of elements and each element in `v1` is equal to the corresponding element in `v2`.
`<, <=, >, >=`	Have their normal meanings using dictionary ordering.

```
vector<int> i(10);
vector<int> j(5, 3);

cout << "First element in i: " << i[0] << endl;
cout << "First element in j: " << j[0] << endl;

j = i;
i[0] = 1;

cout << "First element in i: " << i[0] << endl;
cout << "First element in j: " << j[0] << endl;
```

output:

```
First element in i: 0
First element in j: 3
First element in i: 1
First element in j: 0
```

Using `push_back` member function

To store a new value in a `vector`, you should use the `push_back` member function. This function accepts a value as an argument and store it in a new element placed at the end of the vector. The value is "pushed" at the "back" of the `vector`. The memory for that element is arranged internally for the `vector` without you knowing it. For example,

```
vector<int> x;
x.push_back(12);
```

Q: what happens?

A: This statement creates a new element holding 12 and places it at the end of `x`.

With the introduction of `string` and `vector`, we can easily store words from the standard input into a `vector` container and process these words upon request. For example, we can ask users to input a few words, store them in a `vector` and parse and print those words that are longer than 4 characters.

```
I think this is my mouse
^Z
The words longer than 4 characters are: think mouse
```

VectorStringEx.cpp

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main()
{
    string word;
    vector<string> text;
    while (cin >> word) // CTRL+D to end loop
        text.push_back(word);
    for (auto s : text)
        if (s.size() > 4) cout << s << endl;
    return 0;
}
```

3.3.3 Other `vector` Operations

We can access the elements of a `vector`:

vecEx1.cpp

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
    vector<int> v{ 1, 2, 3 };
    for (auto &i : v) // note: i is a reference
        i *= i; // square the element value
    for (auto i : v) // for each element in v
        cout << i << " "; // print the element
    cout << endl;
    return 0;
}
```

```
1 4 9
```

Quick Check on Concept: What is the difference between `for (auto &i : v)` and `for (auto i : v)`?

```
for (auto &i : v)
```

We define our control variable, `i`, as a reference so that we can use `i` to assign new values to the elements in `v`.

```
for (auto i : v)
```

Control variable, `i`, is a copy of an element in `v`. Any change in `i` will NOT affect the elements in `v`. Thus, we use this kind of `range for` for **read only access** in a container.

<u>Subscript Operator `[]`</u>

We can obtain a given element in `vector` using the subscript operator `[]`. Subscripts for `vector` start at 0 (a typical C/C++ convention). For example, the previous code can now be modified as:

<u>vecEx2.cpp</u>

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
    vector<int> v{ 1, 2, 3 };
    for (decltype(v.size()) idx = 0; idx != v.size(); ++idx){
        v[idx] = v[idx] * v[idx];
        cout << v[idx] << " ";
    }
    cout << endl;
    return 0;
}
```

```
1 4 9
```

<u>Exercise 3.3 In-class Coding Exercise</u>

<u>Ex33.cpp</u>

Define and initialize a vector with 10 elements of 1 and print the contents. Modify all the even indices in the vector to 0 and print the modified contents. A sample output looks like:

```
The original elements in the vector container are:  1 1 1 1 1 1 1 1 1 1
The modified elements in the vector container are:  0 1 0 1 0 1 0 1 0 1
```

(Answer)

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> ivec(10,1);
    cout << "The original elements in the vector container are: "
         << " ";
    for (auto i: ivec)
        cout << i << " ";
    cout << endl;
    cout << "The modified elements in the vector container are: "
         << " ";
    for (decltype(ivec.size()) ix = 0; ix != ivec.size(); ++ix){
        if (ix % 2 == 0) ivec[ix] = 0;
        cout << ivec[ix] << " "; // print the element
    }
    cout << endl;
    return 0;
}
```

More vector functions exist[9], such as `capacity()`.

```
#include <iostream>
#include <vector>

using namespace std;

int main ()
{
    vector<int> i(10);
    cout << i.capacity() << endl;

    i.push_back(7);
    cout << i.capacity() << endl;

    vector<int> j;
    cout << j.capacity() << endl;
    j.push_back(7);
    cout << j.capacity() << endl;
    j.push_back(7);
```

[9] Manual for vectors URL: https://www.cplusplus.com/reference/vector/vector/

```
    cout << j.capacity() << endl;
    j.push_back(7);
    cout << j.capacity() << endl;
    j.push_back(7);
    cout << j.capacity() << endl;
    j.push_back(7);
    cout << j.capacity() << endl;
}
```

From this example, you can see the memory mamangement under the hood. Note that reallocation of memory is a very time consuming task! That is why you might have seen the size doubling when the container is full.